Dynamic Streams for Efficient Communications Between Migrating Processes in a Cluster

Pascal Gallard, Christine Morin

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

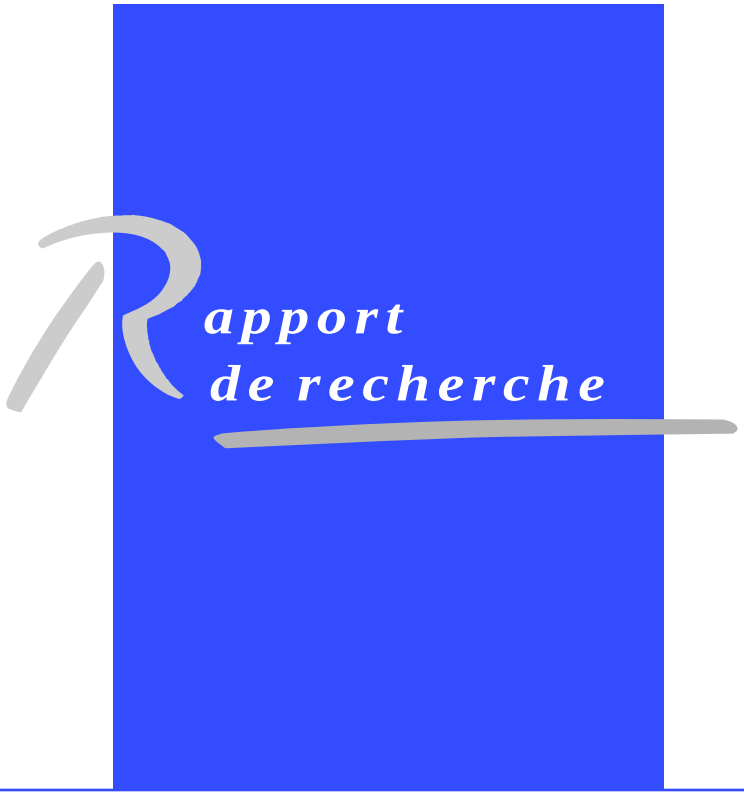
Dynamic Streams for Efficient Communications Between Migrating Processes in a Cluster

Pascal Gallard and Christine Morin

N°4987

Novembre 2003

THÈME 1



ISSN 0249-6399 ISRN INRIA/RR--4987--FR+ENG

Dynamic Streams for Efficient Communications Between Migrating Processes in a Cluster

Pascal Gallard and Christine Morin

Thème 1 — Réseaux et systèmes
Projet PARIS

Rapport de recherche n ° 4987 — Novembre 2003 — 17 pages

Abstract: This paper presents a communication system designed to allow efficient process migration in a cluster. The proposed system is generic enough to allow the migration of any kind of stream: socket, pipe, char devices. Communicating processes using IP or Unix sockets are transparently migrated with our mechanisms and they can still efficiently communicate after migration. The designed communication system is implemented as part of Kerrighed, a single system image operating system for a cluster based on Linux. Preliminary performance results are presented.

Key-words: Network communications, Stream migration, Process migration, MPI, Cluster computing

(Résumé : tsvp)

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : 02 99 84 71 00 - International : +33 2 99 84 71 00
Télécopie : 02 99 84 71 71 - International : +33 2 99 84 71 71

Flux dynamique pour la migration efficace de processus communiquant au sein d'une grappe de calculateurs

Résumé : Cet article propose un système de communication conçu afin de permettre la migration efficace de processus communiquant au sein d'une grappe de calculateurs. L'architecture proposé est suffisament générique pour permettre la migration de tous types de flux: socket, tube, périphérique de type caractère. Les processus communiquant par sockets IP ou Unix sont déplacés de manière transparente à l'aide de notre système, et les communications restent efficaces après la migration. L'architecture proposée à été développée dans Kerrighed, un système d'exploitation à image unique pour grappe de calculateurs basé sur Linux. Une évaluation préléminaire des performances est présentée.

Mots-clé : Communication réseaux, Migration de flux, Migration de processus, MPI, grappe de calculateurs

1 Introduction

Clusters are now more and more widely used as an alternative to parallel computers as their low price and the performance of micro-processors make them really attractive for the execution of scientific applications or as data servers.

A parallel application is executed on a cluster as a set of processes which are spread among the cluster nodes. In such applications, processes may communicate and exchange data with each other. In a traditional Unix operating system, communication tools can be streams like `pipe` or `socket` for example. For load-balancing purpose, a process may be migrated from one node to another node. If this process communicates, special tools must be used in order to allow high performance communication after the migration.

This paper presents a new communication layer for efficient migration of communicating processes. The design of this communication layer assumes that processes migrate inside the cluster and do not communicate with processes running outside the cluster. In the Kerrighed operating system [18, 10], depending on the load-balancing policy, processes may migrate at any time.

The remainder of this paper is organized as follows. Related work is reviewed in Section 2. Section 3 describes the dynamic stream service providing the dynamic stream abstraction and Kerrighed sockets. Section 4 shows how the dynamic stream service can be used to implement pipes and distributed Unix sockets. In Section 5 we provide an overview of the low-level communication service on which the dynamic stream service is based. Section 6 presents performance results obtained with the Kerrighed prototype. Conclusions and future work are presented in Section 7.

2 Background

The problem of migrating a communicating process is difficult and this explains why several systems, such as Condor [9], provide process migration only for non communicating processes.

The MOSIX system [2, 1] uses *deputy* mechanisms in order to allow the migration of a communicating process. When a process migrates (from a *home*-node to a *host*-node), a link is created between this process (on the *host*-node) and the *deputy* (on the *home*-node). Every communication from/to this process is transmitted to the *deputy* that acts as the process. In this way, migrated processes are not able to communicate directly with other processes and thus communication performance decreases after a migration. The Sprite Network operating system [5, 6] uses similar mechanisms in order to forward kernel calls whose results are machine-dependent.

Several other attempts to provide communication migration have been done at different levels of the OSI/ISO model.

PVM [17] and MPI [7] applications are good examples of communicating programs. Several works like MPVM [3] or CoCheck [14] allow the migration of processes. However, these middlewares are not transparent for applications.

Mobile-TCP [12] provides a migration mechanism in the TCP protocol layer using a *virtual port* linked to the real TCP socket.

Mobility is one of the main features of IPv6 [11] but communications can migrate only if the IP address migrates. In this case, one process must be attached to one IP address and each host must have several IP addresses (one for each running communicating process). Even in this case, only one kind of communication tools (inet sockets) can migrate.

Another case of communication migration is detailed in M-TCP [16] where a running client, outside of the cluster, can communicate with a server through an access point. If processes on servers migrate, access points can hide the communication changes.

None of these proposals offers a generic and efficient mechanism for migrating streams in a cluster allowing a migrating process to use all standard communication tools of a Unix system.

We want to avoid message forwarding between cluster nodes when processes migrate. We propose a generic approach in order to provide standard local communication tools like `pipe`, `socket` and *char*-devices compliant with process migration (decided for load-balancing reasons or due to configuration changes in the cluster – addition/eviction of nodes). This approach has been implemented as part of the Kerrighed project at the operating system level and in this way provides full migration transparency to communicating applications.

3 Dynamic Streams

Our work aims at providing standard communication interfaces such as Unix sockets or pipes to migrating processes in a cluster. Migrating a process should not alter the performance of its communications with other processes.

A communication comprises two distinct aspects: firstly a binary stream from a node A to a node B, and secondly a set of meta-data providing information about the state of the stream and how to handle it. Our architecture is based on this idea.

We propose the concept of dynamic streams on which standard communication interfaces are built. We call the extremities of these streams "KerNet sockets" and these can be migrated inside the cluster. Dynamic streams and KerNet sockets are implemented on top of a portable high performance communication system providing a send/receive interface to transfer data between different nodes in a cluster.

The proposed architecture is depicted in Figure 1. The low-level point-to-point communication service is based either on device drivers (such as Myrinet), the generic network device in the Linux kernel (netdevice) or a high-level communication protocol (such as TCP/IP). On top of the low level point-to-point layer, we provide three kinds of dynamic streams: di-

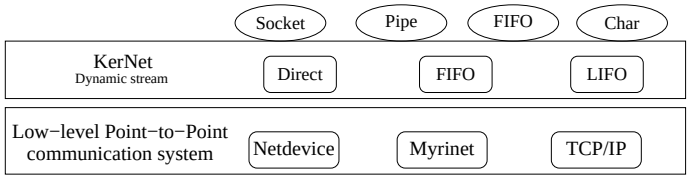


Figure 1: Kerrighed network stack

rect, FIFO and LIFO streams. Dynamic streams are specialized by interfaces. We use these dynamic streams, implemented by the *KerNet layer* to offer dynamic versions of standard Unix stream interfaces (inet/unix sockets, pipe, ...).

The KerNet layer implements the abstraction of dynamic stream and KerNet sockets. It is a distributed service which provides global stream management cluster wide. In the remainder of this paper, we focus on the design and implementation of the KerNet layer and of the Unix socket interface and pipes. The low-level point-to-point communication system is only briefly described.

3.1 Dynamic Stream Service

All systems for clusters rely on a low-level point-to-point communication system. Most of the time, such a system provides properties such as reliability, high-performance and message ordering. So, when no migration occurs, applications should be able to take advantage of these properties.

In Kerrighed, we designed a low-level communication system which is reliable and keeps the message sending order between two nodes. This system is described in Section 5.

We define a KerNet dynamic stream as an abstract stream with two or more defined KerNet sockets and with no node specified. When needed, a KerNet socket is temporarily attached to a node. For example, if two KerNet sockets are attached, send/receive operations can occur.

A KerNet dynamic stream is mainly defined by several parameters:

- **Type of stream**: It specifies how data is transfered using the dynamic stream. A stream can be:
  - DIRECT for one to one communication, such as socket based communication,
  - FIFO or LIFO for streams with several readers and writers.
- **Number of sockets**: It specifies the total number of available sockets for the stream. Depending on the stream type, this value may increase, decrease or be constant.
- **Number of connected sockets**: It specifies the current number of attached sockets.

- **Data filter**: It allows modification of all data transmitted with the stream (in order to have cryptography, backup...).

Streams are managed by a set of stream managers, one executing on each cluster node. Kernel data structures related to dynamic streams are kept in a global directory which is distributed on cluster nodes. Data such as the *node location* of a KerNet socket, are updated on all the nodes acting on the stream.

3.2 KerNet Sockets

The KerNet service provides a simple interface to allow upper software layers implementing a standard communication interface to manage KerNet sockets:

- `create`/`destroy` a stream,
- `attach`: To get an available KerNet socket (if possible),
- `suspend`: To unattach temporarily a socket (and to give a handle in order to be able to reclaim the KerNet socket later),
- `wakeup`: To reactivate a previously suspended KerNet socket,
- `unattach`: To release an attached KerNet socket,
- `wait`: To wait for the stream to be ready to be used (all required attachments completed).

KerNet provides two other functions (`send`, `recv`) for I/O operations.

The dynamic stream service is in charge of allocating KerNet sockets when there are needed, and of keeping track of these KerNet sockets. When the state of one KerNet socket changes, the stream's manager takes part in this change and updates the other KerNet sockets related to the stream. With this mechanism, each KerNet socket has got the address of each corresponding socket's node in a map. In this way, two sockets can always communicate in the most efficient way (using this map).

At the end of a connection, a process is unattached from the stream. Depending on the stream type, the stream may be closed.

3.3 Example of Utilization of the KerNet API in the OS

Let us consider two kernel processes (*P1* and *P2*) communicating with each other using a `DIRECT` dynamic stream. Later we start a third process (*P3*) for migration purpose. They execute the program depicted in Figure 2.

```
     P1                                P2
(1)  stream = create(DIRECT, 2);
(2)  socket1 = attach(stream);         socket2 = attach(stream);
(3)  wait(stream);                     wait(stream);
(4)  ch = recv(socket1);               send(socket2, ch);
(5)  ...                               ...
(6)                                    sec2 = suspend(socket2);
(7)                                    exit

                                       P3
(8)                                    socket3 = wakeup(stream,
                                                        sec2);
(9)  send(socket1, ch);                ch = recv(socket3);
```

Figure 2: Example of KerNet level code

Initialization of a KerNet stream: **P1** creates the stream (the stream's type and the number of socket in the stream are given in parameters) and requests a KerNet socket. Next **P1** waits for its stream to be ready, that is to say the two sockets to be attached. Assuming **P2** is running after the stream creation and has the stream identifier, it can get a KerNet socket, and then, wait for the correct state of the stream. The stream's manager sends the acknowledgement to all waiting KerNet sockets and provides the physical address of the other socket. With such information, KerNet sockets can communicate directly and *send/receive* communication can occur efficiently.

Migrating a process using KerNet streams: When a process migration is needed, the KerNet socket is suspended on the departure node and re-attached on the arrival node. In our example, **P2** suspends the socket, which is later re-attached when **P3** executes. The dynamic stream service is in charge of ensuring that no message (or message piece) is corrupted or lost between the suspend and re-attach time. The stream manager updates the other KerNet socket so that it stops its communication until it receives new information from the stream manager. When the suspended socket is activated again, its new location is sent to the other KerNet socket and direct communication between the two KerNet sockets is restarted. If a migration starts in the same time that a message is sent, the first sending attempt is discarded and we rely on the low-level communication layer in order to keep the message until a new destination is defined. When the location is updated, the message is sent to the new correct node.

4 Implementation of Standard Communication Interfaces using Dynamic Streams

Obviously, standard distributed applications do not use KerNet sockets. In order to create a standard environment based on dynamic streams and KerNet sockets, an interface layer is implemented at kernel level (see Figure 3). Each module of the interface layer implements a standard communication interface relying on the interface of the KerNet service. The main goal of each interface module is to manage the standard communication interface protocol (if needed).

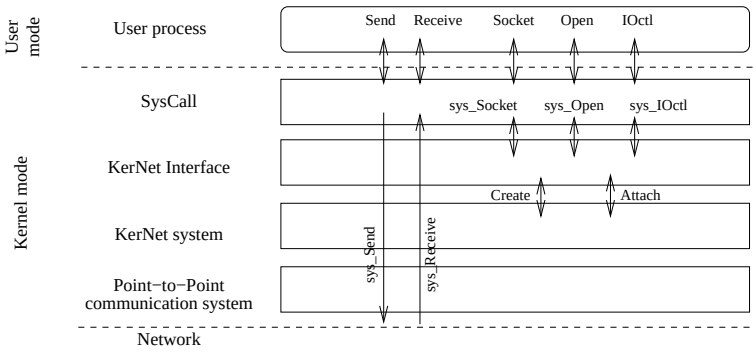


Figure 3: Standard environment based on KerNet sockets

KerNet interfaces are the links between the standard Linux operating system and the Kerrighed dynamic communication service.

The Kerrighed operating system is based on top of a lightly-modified Linux kernel. All the different services, including the communication layer, are implemented as Linux kernel modules.

In Kerrighed operating system, the communication layer is made of two parts. A static high-performance communication system that provides a node to node service. On top of this system, the dynamic stream service manages the migration of streams's interfaces. Finally, the interface service replaces the standard functions for a given communication tool.

In the remainder of this section, we describe two different kinds of KerNet interfaces: the pipe and the Unix socket[15] interfaces on top of the KerNet sockets. We aim at providing transparently to the applications, a distributed version of these communication tools.

4.1 Pipe Communication

`Pipes` belong to the most common and oldest communication tools. Pipes are an easy way to connect a process to a child process. The *pipe* system call creates a pair of file descriptors, one is for reading and the other one is for writing.

In Figure 4, one process **P1** creates a pipe and gets two file descriptors (*fd1* and *fd2*). When **P1** performs a `fork`, the child process **P2** inherits the file descriptors from **P1**. **P1** and **P2** may close one file descriptor and use the other one to communicate.

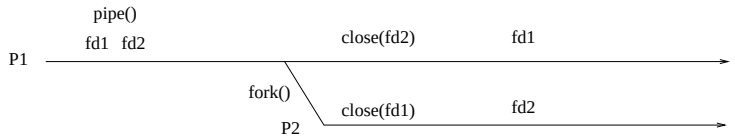


Figure 4: Traditional pipe usage

Since the two extremities of a pipe are created at the same time, there is no special protocol to implement in Kerrighed. In order to implement pipes in our KerNet architecture, we just have to create a stream with two KerNet sockets. Each KerNet socket is attached to each file descriptor (as shown in Figure 5).

```
      P1                                    P2
(1)   pipe(fd[2])
(1.1)     |  stream = create(DIRECT, 2);
(1.2)     |  fd[1] = attach(stream);
(1.3)     |  fd[2] = attach(stream);
          +----
(2)   fork()
(3)   close(fd2)                            close(fd1)
(4)   write(fd1,...)                        read(fd2,...)
```

Figure 5: Basic pipe implementation

4.2 Unix Sockets

In the standard Linux kernel, `Unix sockets` are as simple as unqueuing some packets from the sending socket and queuing them in the receiving socket. In this case the (physical) shared memory provides the operating system access to the system structures of the two sockets. In the same way, the protocol management can be done easily. Obviously, in our architecture, the operating system of one node may not have access to the data structure

of the other socket. Based on the KerNet services, the KerNet Unix sockets interface must manage the standard Unix sockets communication protocol.

When a new interface is defined, a corresponding class of streams is registered in the dynamic stream service. A class of streams is a set of streams that share the same properties. This registering step defines general properties of streams associated to this interface (stream type, number of sockets, ...) and returns a stream class descriptor.

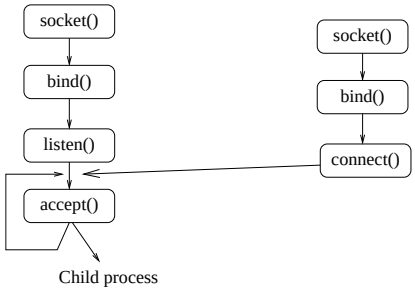


Figure 6: Server and client algorithm for Unix socket usage

Figure 6 shows the traditional usage of a Unix socket. When a process makes an `accept` on a Unix socket, the Unix socket interface creates a new stream and attaches the first KerNet socket and waits that the stream allocates the other KerNet socket (as in **P1** in Figure 2). When another process (maybe on another node) executes a `connect` on the Unix socket, an attach attempt is made (process **P2** in Figure 2). On success, the stream is completed and the two KerNet sockets (and by this way the two Unix sockets) can communicate directly.

The `accept`/`connect` example is a good representation of how we implement Unix sockets. With the same approach we have designed and implemented other standard socket functions like `poll` (in order to be able to use the `select` system call), `listen`, and so on.

The `send` and `receive` functions are directly mapped on to the `send` and `receive` KerNet socket functions.

Based on this stream interface, we can make other interfaces for `inet socket` communication and even access to `char` devices.

When a migration occurs (decided by the global scheduler or by the program itself), the migration service calls the `suspend` function and attaches the socket on the new node (such as **P3** in Figure 2).

5 Low-Level Communication Layer

The KerNet architecture is designed for a distributed operating system such as Kerrighed. For this reason, it was natural to build KerNet on top of the low-level communication layer

provided by Kerrighed and called Gimli/Gloïn. In the following, we highlight some features of this communication layer.

- **High performance**: This communication system provides a low latency and high bandwith communication service for kernel to kernel communications.
- **Network independence**: There is no dependency between this communication layer and some special network devices.
- **Reliability**: Every sent message is delivered without any change in the message.
- **Simple interface**: This communication service is based on the channel idea, and provides a very basic `send`/`receive` interface.

Kerrighed low level point to point communication system is divided in two layers: Gimli and Gloïn. Gimli provides a kernel level API on which all Kerrighed services rely, in particular KerNet. Gloïn is in charge of the communication reliability management (error control, packets retransmission). Finally, Gimli provides a way to classify the messages according to some profile, a message being characterized by a channel and a type. If a message cannot be delivered, an error is returned to the calling service (KerNet in our case).

While Gimli is independent from the network technology, Gloïn is interfaced with different communication systems: the standard TCP/IP stack, netdevice (a generic communication layer on top of the standard network access interface) or dedicated high performance communication systems such as Gamma [4]. Drivers need not be modified. However, the architecture has been designed to allow the exploitation of specialized drivers for performance reasons. KerNet being based on Gimli is hence portable on any networking technology (Myrinet, Gigabit Ethernet, ...). Its performance is directly dependent on the Gloïn driver used and on the performance of the underlying network.

The interaction between Gimli/Gloïn and the KerNet service is mainly done by the `KerNet_Send` function. Basically, this function tries to send the message for ever passing the socket's map as node destination to the `gimli_send` function (see Figure 7, KerNet layer). When the the receiver migrates, `gimli_send` failed. The `gimli_send` will success when the map will be updated at the end of the migration (with the new location). Until this event, the previous location discards the unwanted messages.

6 Performance Evaluation

Performance evaluations have been carried out to show that in any process configuration, KerNet is able to provide the best performance as possible. These performances should be outperformed by our low-level communication layer. In the last part of our evaluation, we use a standard communication system such as MPI on KerNet architecture.

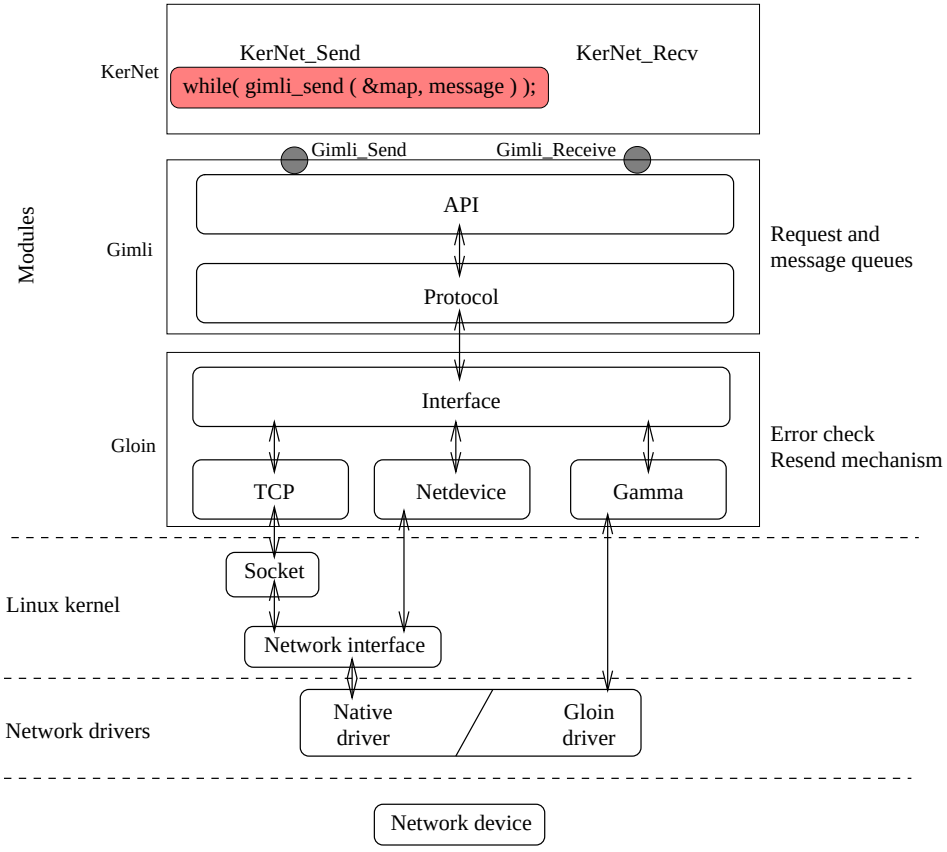


Figure 7: Low-Level communications service

Our test application is the NetPIPE [13] application. This benchmark is a ping-pong application between two processes, with several packet sizes. In order to place these processes, we have two cases:

- On the same node,
- On separate nodes.

When the two processes are on the same node, the inter-connection networks are:

- inet loopback interface: TCP/IP socket (*vanilla*-TCP),
- shared memory: Unix socket (*vanilla*-Unix),
- low-level loopback: KerNet TCP (TCP-like),
- low-level loopback: KerNet Unix (distributed unix socket).

When the two processes are on separate nodes, the inter-connection networks are (with the same interfaces):

- FastEthernet,
- Gigabit Ethernet.

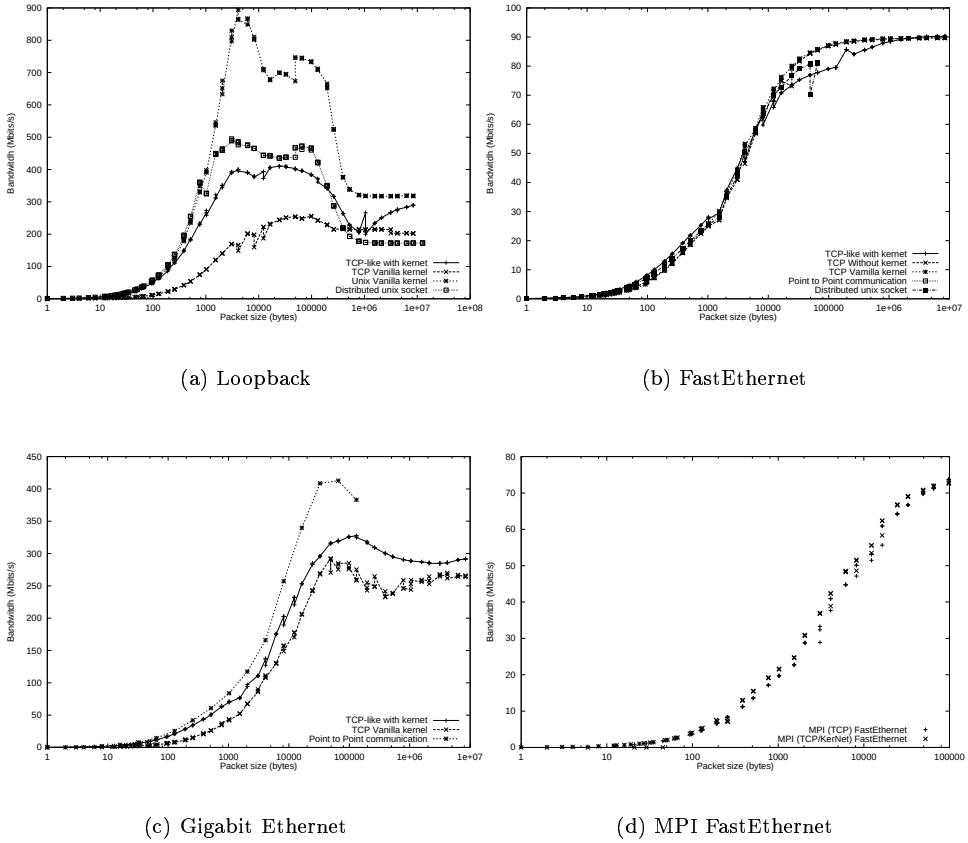
The nodes in our cluster are *Pentium III* (500MHz, 512KB cache) with 512MB of memory. The Kerrighed system used is an enhanced 2.2.13 Linux kernel.

In addition, we provide in all cases, the performance of our point-to-point communication layer (without any dynamic functionality). We must notice that these measures represent point-to-point communications from kernel to kernel with buffers physically allocated in memory. In this case, buffers are in a contiguous memory area and their size is lower than 128KB. In all cases, the point-to-point low-level performance is a maximum for our KerNet stream.

When two communicating processes are on the same node, dynamic streams outperform the standard TCP sockets. This is mainly due to the small network stack in KerNet: the IP stack provides some network services which are never used in a cluster or already performed by the Kerrighed high-performance communication layer. However we do not reach (on a single node) the performance of a Unix socket. This is mainly due to the design of the low-level communication layer which has been designed for inter-node communications without any optimization for local communications.

When the two communicating processes are not on the same node, KerNet outperforms TCP sockets again. The reasons are the same as above.

We notice that the interfaces have a low impact on the dynamic streams: Unix sockets and TCP sockets have nearly the same results. With a *FastEthernet* network, the KerNet



(a) Loopback

(b) FastEthernet

(c) Gigabit Ethernet

(d) MPI FastEthernet

Figure 8: Throughput of several communication systems

dynamic stream bandwidths are nearly the same as those of the point-to-point low-level communication system. With a *Gigabit Ethernet* network, we notice a difference between our ping-pong application and the point-to-point low-level communication system. With our current low-level communication system, we have several copies in the data path:

- One from the network card to packet buffer,
- One from the packet buffer to the message (in kernel space),
- Finally one to the user-space buffer provided by the application.

With a more efficient network such as the Gigabit one, the last copy is enough to justify the difference between the low-level communication layer and the user-space communication layer.

Finally, we made some evaluations of a standard communication middleware such as MPI (MPICH 1.2.5.6) [8]. Figure 8(d) shows a comparison between an MPI version of the NetPIPE application running on a standard Linux system, and the same application running on Kerrighed. In this case, both MPICH and NetPIPE are unmodified. With Kerrighed, we set MPI in order to run on one node (the nodes configuration file includes a single entry: `localhost`) and we rely on the Kerrighed global process management service[19] in order to place (and migrate) processes on distinct nodes in the cluster. With the dynamic streams service, we ensure that processes are kept connected. The evaluation shows that there is no overhead added by our communication architecture.

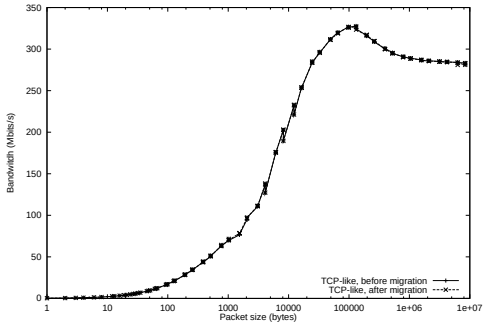


Figure 9: Migration cost on a Gigabit network

Other experiments have been performed to evaluate the impact of a migration on the dynamic stream performance. The NetPIPE application for TCP (NPtcp) has been modified to trigger a KerNet socket migration. Figure 9 shows that there is no overhead after a migration of a communicating process.

7 Conclusion

In this paper we have described the design and implementation of a distributed service allowing efficient execution of communicating processes after migration. We have introduced the concept of dynamic streams and migratable sockets. We have shown by means of the example of Unix sockets how standard communication interfaces can take advantage of these concepts.

The proposed scheme has been implemented in the Kerrighed operating system. Processes of parallel applications based on the message passing communication paradigm such

as MPI applications can be migrated without any performance degradation due to communications taking place after migration. We currently work on experimentations with industrial applications.

In future work on dynamic streams we plan to provide other standard stream communication interfaces like `FIFO` and access to `char` devices. We also plan to study fault-tolerance issues in the framework of the design and implementation of checkpoint/restart mechanisms for parallel applications in the Kerrighed cluster operating system. In addition, we want a more specific design of the low-level communication service in order to improve the overall KerNet performance (by removing at least one copy).

References

[1] Lior Amar, Amnon Barak, and Amnon Shiloh. The MOSIX Parallel I/O System for Scalable I/O Performance. In *Proc. 14-th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2002)*, pages 495–500, Cambridge, MA, Nov 2002.

[2] Amnon Barak, Shai Guday, and Richard G. Wheeler. *The MOSIX Distributed Operating System, Load Balancing for UNIX*, volume 672 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.

[3] Jeremy Casas, Dan L. Clark, Ravi Konuru, Steve W. Otto, Robert M. Prouty, and Jonathan Walpole. MPVM: A migration transparent version of PVM. *Usenix Computing Systems*, 8(2):171–216, 1995.

[4] Giuseppe Ciaccio. Optimal communication performance on Fast Ethernet with GAMMA. In *IPPS/SPDP Workshops*, volume LNCS 1388, pages 534–548. Springer-Verlag, 1998.

[5] F. Douglis and J. Ousterhout. Transparent process migration: Design alternatives and the Sprite implementation. *Software–Practice & Experience*, 21(8), August 1991.

[6] Frederick Douglis. *Transparent Process Migration in the Sprite Operating System*. PhD thesis, U.C. Berkeley, September 1990. Report UCB/CSD 90/598.

[7] Message Passing Interface Forum. MPI: A message-passing interface standard. Technical Report UT-CS-94-230, University of Tennessee, 1994.

[8] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

[9] Michael Litzkow, Todd Tannenbaum, Jim Basney, and Miron Livny. Checkpoint and Migration of UNIX Processes in the Condor distributed processing system. Technical Report 1346, University of Wisconsin-Madison Computer Sciences, April 1997.

[10] Christine Morin, Pascal Gallard, Renaud Lottiaux, and Geoffroy Vallée. Towards an efficient single single system image cluster operating system. In *Proc. 5-th International Conference on Algorithms and Architectures for Parallel Processing ICA3PP*, pages 370–377, Beijing, China, October 2002.

[11] Charles E. Perkins and David B. Johnson. Mobility Support in IPv6. In *Mobile Computing and Networking*, pages 27–37, 1996.

[12] Xun Qu, Jeffrey Xu Yu, and Richard P. Brent. A mobile TCP socket. Technical Report TR-CS-97-08, Australian National University, Canberra 0200 ACT, Australia, 1997.

[13] Q. Snell, A. Mikler, and J. Gustafson. NetPIPE: A Network Protocol Independent Performace Evaluator. In *IASTED International Conference on Intelligent Information Management and Systems*, June 1996.

[14] Georg Stellner. CoCheck: Checkpointing and Process Migration for MPI. In *Proceedings of the 10th International Parallel Processing Symposium (IPPS'96)*, Honolulu, Hawaii, April 1996.

[15] Richard Stevens. *Unix Network Programming*, volume 1-2. Prentice Hall, 1990.

[16] F. Sultan, K. Srinivasan, D. Iyer, and L. Iftode. Migratory TCP: Highly Available Internet Services using Connection Migration. In *Proceedings of The 22nd International Conference on Distributed Computing Systems (ICDCS)*, July 2002.

[17] V. S. Sunderam. PVM: a framework for parallel distributed computing. *Concurrency, Practice and Experience*, 2(4):315–340, 1990.

[18] Geoffroy Vallée, Renaud Lottiaux, Louis Rilling, Jean-Yves Berthou, Ivan Dutka-Malhen, and Christine Morin. A case for single system image cluster operating systems: Kerrighed approach. *Parallel Processing Letters*, 13(2), June 2003.

[19] Geoffroy Vallée, Christine Morin, Jean-Yves Berthou, and Louis Rilling. A new approach to configurable dynamic scheduling in clusters based on single system image technologies. In *Proc. of the 17th International Parallel and Distributed Processing Symposium (IPDPS 2003)*, page 91, Nice, April 2003. IEEE.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399